Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Lab 10: ~~Tuesday 4/11~~ May 1 @ 11:59 pm
  - Homework 11: ~~this Friday~~ May 1 @ 11:59 pm
  - Assignment 4: ~~this Friday~~ May 1 @ 11:59 pm
    - Support video and slides on Canvas + Solutions for Labs 8 and 9
  - Midterm Question Reattempts: Monday 4/17 @ 11:59 pm
    - up to **7 points** back
    - Please use GradeScope's Regrade Requests for each question **individually**

Previous Lecture …

Dynamic Programming

- A recipe
- Examples:
  - Computing the n^{th} Fibonacci Number
  - Unbounded Knapsack

This Lecture ...

Dynamic Programming: Typical question in **coding interviews**!

- More Examples:
  - 0/1 Knapsack
  - Change Making
  - Subset Sum
  - Edit Distance
  - Longest Common Subsequence
  - Reinforcement Learning

Dynamic Programming: a recipe

- What is the **first decision** to make to solve the problem?
- What **subproblem**(s) emerge out of the that first decision?
- Must **wait** for subproblem solutions to make first decision
- start with a recursive solution
- if inefficient, do you have **overlapping** subproblems?
- identify the **unique** subproblems
- Allocate an **array** to hold their solutions
- solve them from **bottom-up** smaller to larger
- Optimize space if possible

Dynamic Programming: a recipe

- How to **combine** subproblem solutions to a problem’s solution?
- What are the **unique** subproblems?

Example 3: The 0/1 knapsack problem

- a **finite** set of items each with a weight and value
  - Two **choices** for each item:
    - goes in the knapsack or
    - left out

weight:	6	3	4	2
value:	30	14	16	9

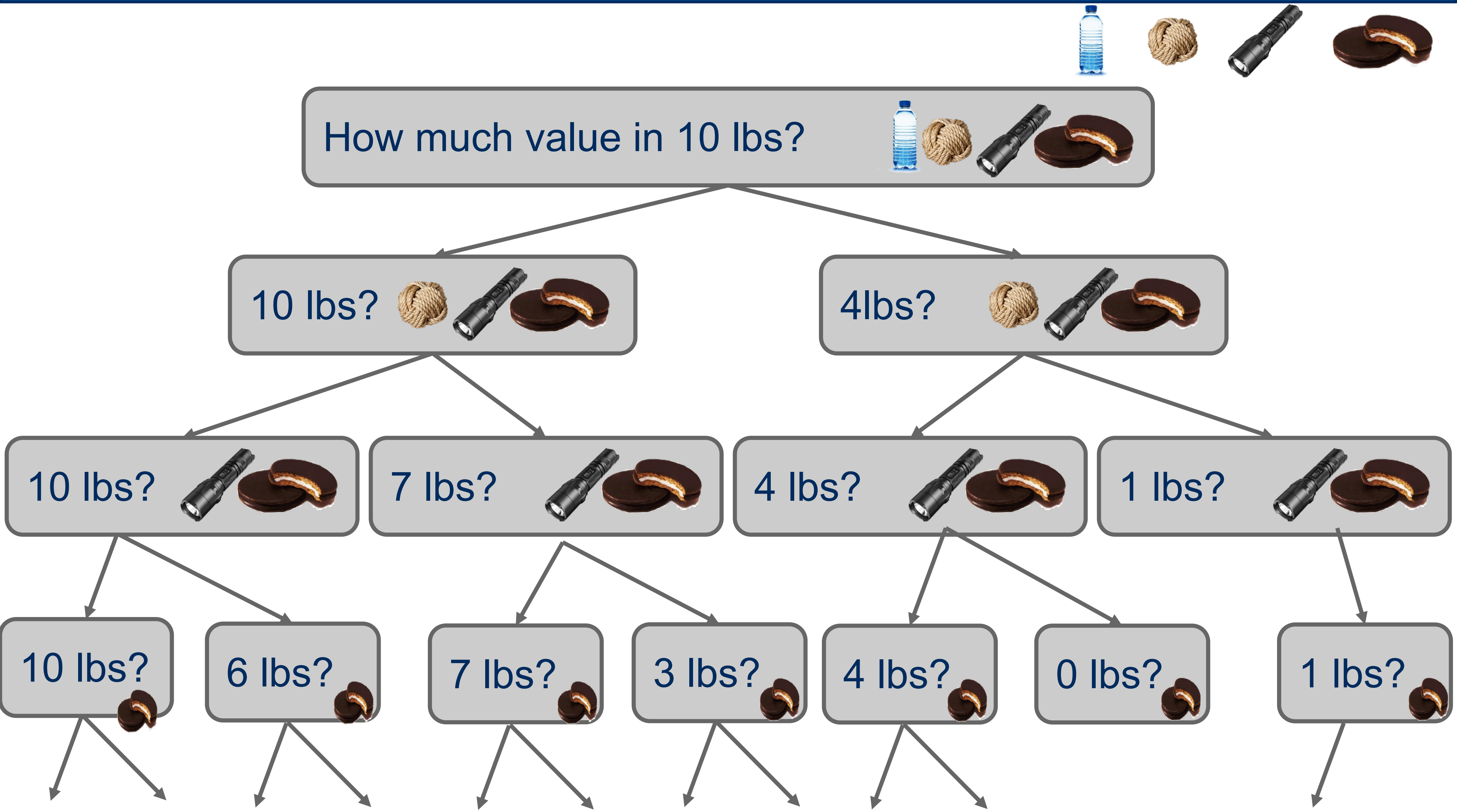
- What would be our **first decision**?
  - to place or not the first item (or last item)

10 lb. capacity

- What **suproblems** emerge?
  - item placed → one less item and capacity less by item’s weight
  - item not placed → one less item and same capacity
  - which choice to take?
  - do we have to **wait** for both subproblem solutions?

Recursive solution

weight:	6	3	4	2
value:	30	14	16	9



Recursive solution

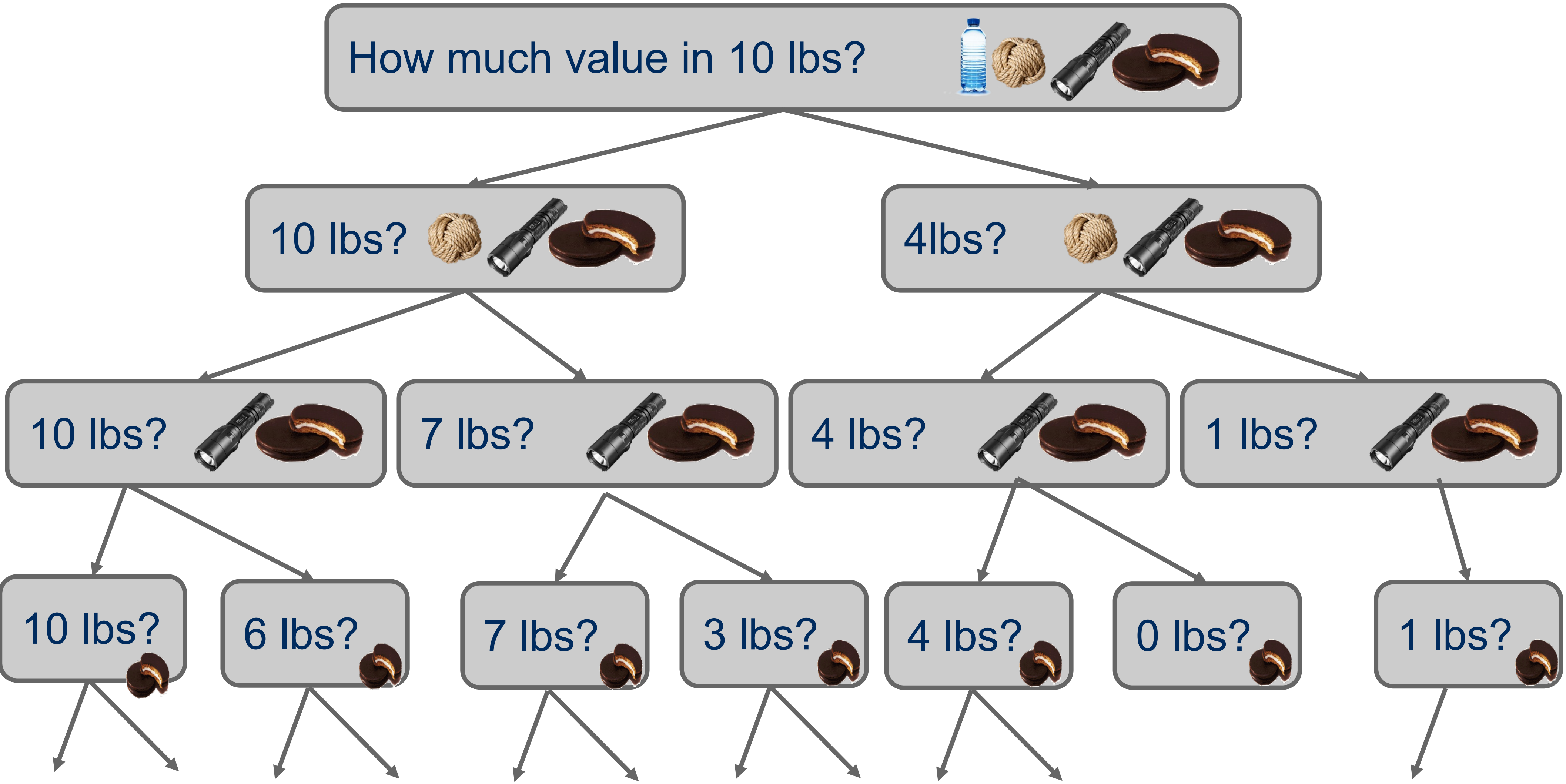
```
int knapSack(int[] wt, int[] val, int L, int n) {
    if (n == 0 || L == 0) { return 0 };
    //try placing the (n-1)st item
    if (wt[n-1] > L) { //cannot place
        return knapSack(wt, val, L, n-1)
    } else {
        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                    knapSack(wt, val, L, n-1)
                  );
    }
}
```

place the item

don't place the item

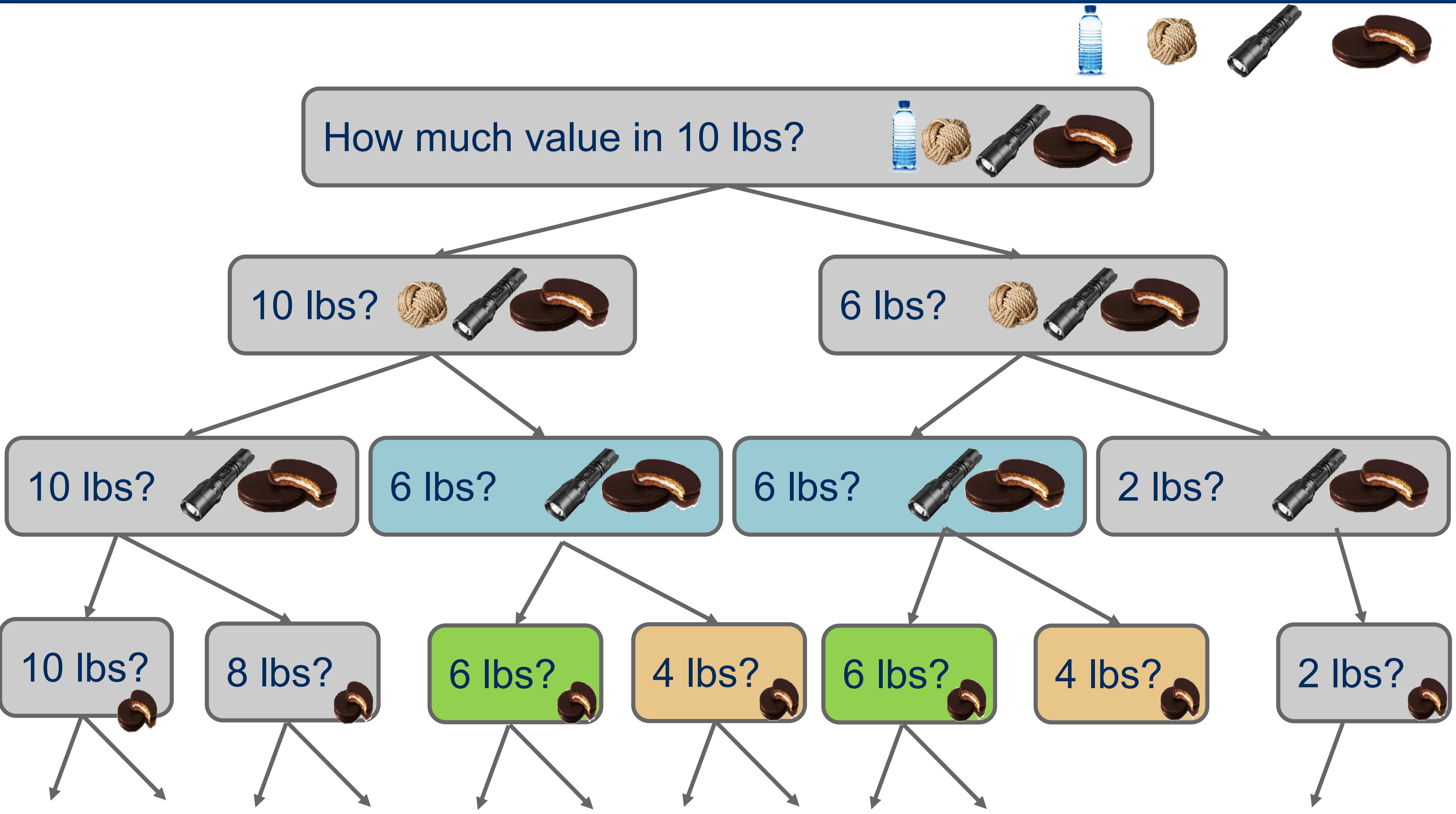
Overlapping Subproblems?

weight:	6	3	4	2
value:	30	14	16	9



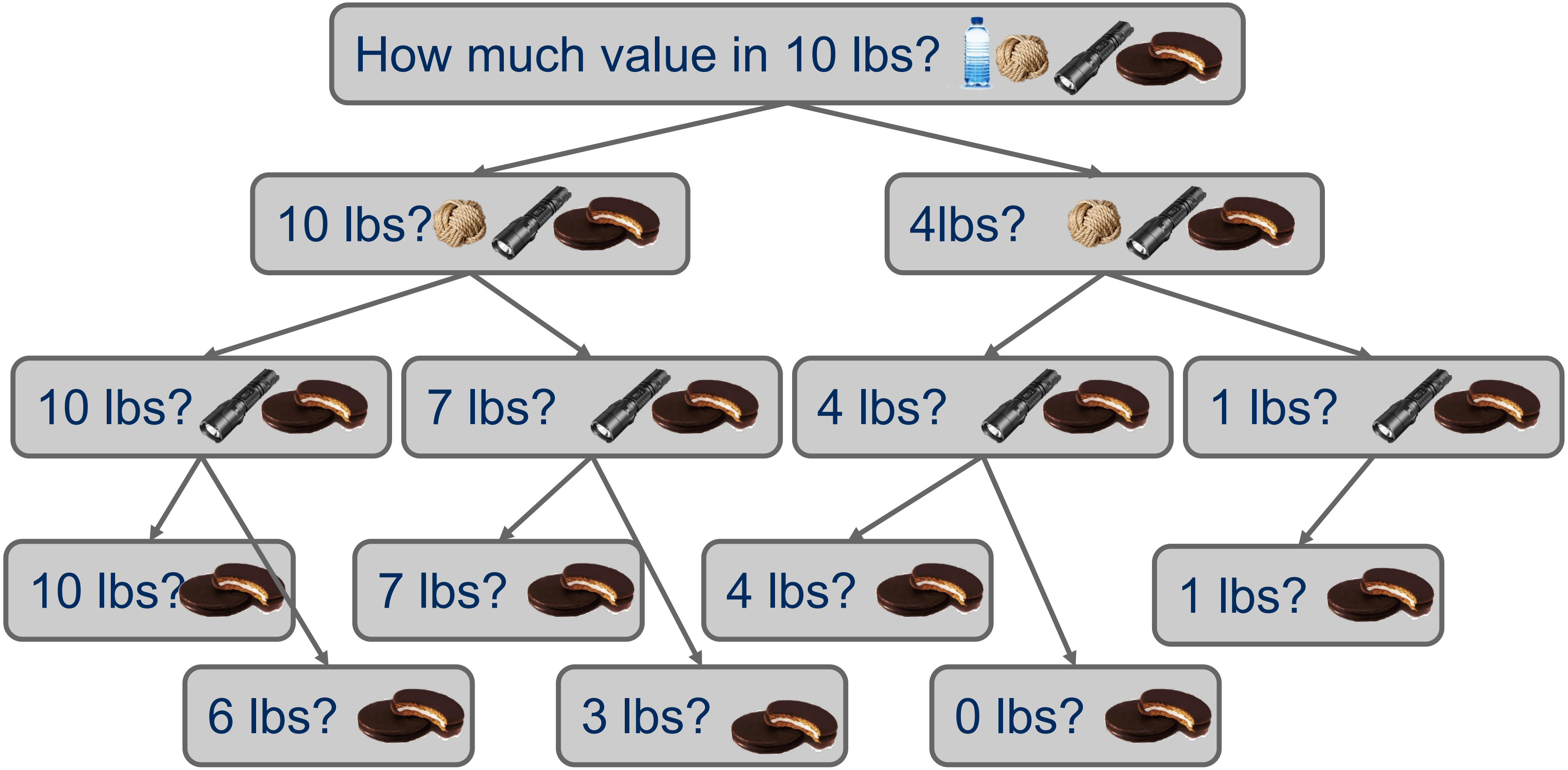
Overlapping Subproblems?

weight:	4	4	2	2
value:	30	14	16	9



Subproblems

- What are the unique subproblems?
- What array should we use to store their solutions?
  - 2-D array!



The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
K[n+1][L+1]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0){ K[i][l] = 0 };
```

place the item

don’t place the item

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**										
2	**0**										
3	**0**										
4	**0**										

K[i][l] is the best (max) value when only the first ***i*** items are available and only ***l*** lbs remain in the knapsack

The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0){ K[i][l] = 0 };
            //try to add item i-1
            else if (wt[i-1] > l){ K[i][l] = K[i-1][l] };
            else {
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]],
                              K[i-1][l]);
            }
        }
    }
    return K[n][L];
}
```

place the item

don't place the item

The 0/1 knapsack dynamic programming solution

subproblem if item *i* placed

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

subproblem if item *i* not placed

i\l	0	1	2	3	4	5	6	7				
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											

\+ value of item *i*

K[i][l] is max of the two subproblems' solutions

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0					
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0										
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0								
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0										
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16						
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0										

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0									

The 0/1 knapsack dynamic programming solution

```
wt  = [ 6,  3,  4,  2 ]
val = [ 30, 14, 16, 9 ]
```

i\l	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	16	30	30	39	44	

Example 4: the change making problem

- What is the **minimum** number of coins needed to make up a given change value **k >= 0**?
- If you were working as a cashier, what would your algorithm be to solve this problem?

This is a *greedy algorithm*

- At each step, the algorithm makes the choice that seems to be best **at the moment**

... But wait ...

- Does our greedy change making algorithm solve the change making problem?
  - For US currency ...
    - yes!
  - But what about a currency composed of
    - pennies (1 cent), thrickels (3 cents), and fourters (4 cents)?
    - What denominations would it pick for k=6?

So what changed about the problem?

- For greedy algorithms to produce optimal results, problems must have two properties:
    - **Optimal substructure**: optimal solution to a subproblem leads to an optimal solution to the overall problem
        - best way to make change for 3 cents → best way to make 6 cents
    - The **greedy choice** property
        - Globally optimal solutions assembled from locally optimal choices
        - K = 6: for US currency, the best overall choice will be to use the biggest coin (nickel)
        - With thrickels/fourters, we can't know until we've looked at all possible breakdowns
- Why is optimal substructure not enough?

So, how can we solve the change making problem optimally?

We will see a dynamic programming algorithm in the recitations

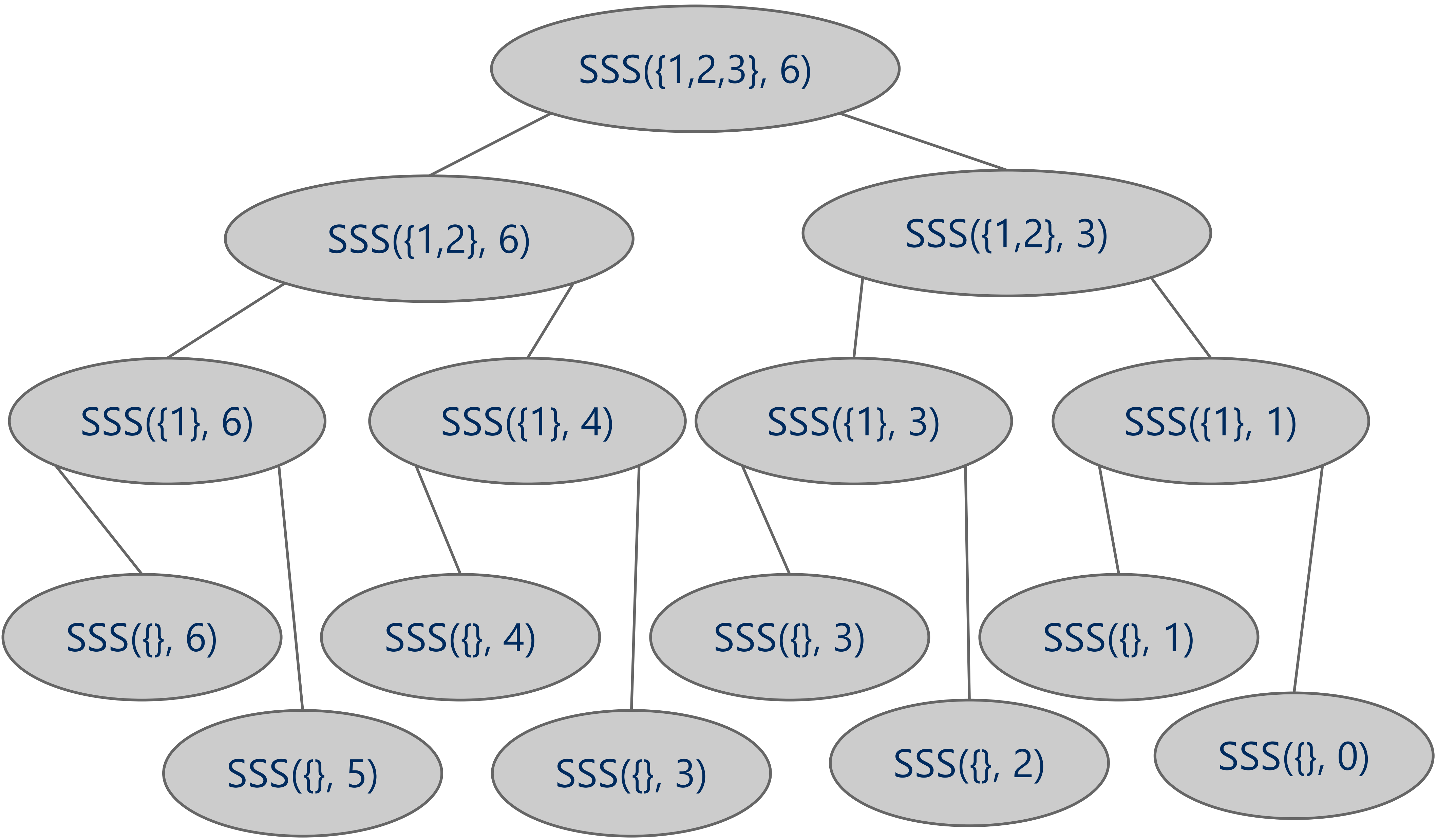
Example 5: Subset sum

- Given a set of **non-negative integers** S and a **target sum** k, is there a **subset** of S that sums to **exactly** k?

Dynamic Programming: a recipe

- Decision: whether last item in input set is in solution subset
  - try both alternatives!
- How to **combine** subproblem solutions to a problem's solution?
  - logical OR (|| in Java)
- What are the **unique** subproblems?

Subset sum calls



Subset sum recursive solution

```
boolean SSS(int set[], int sum, int n) {
    //base cases
    if (sum == 0)
          return true;
    if (sum != 0 && n == 0)
          return false;
    //can we include item n-1?
    if (set[n-1] > sum)
          return SSS(set, sum, n-1);
    //should we include item n-1?
    return SSS(set, sum, n-1) ||
           SSS(set, sum-set[n-1], n-1);
}
```

The Subset Sum dynamic programming solution

```
set  = [1, 2, 3]

subset[n+1][sum+1]
```

i\j	0	1	2	3	4	5	6
0							
1							
2							
3							

subset[i][j] is true iff a subset of the first ***i*** items sums up to ***j***

The Subset Sum dynamic programming solution

```
set  = [1, 2, 3]
```

```
subset[n+1][sum+1]
```

i\j	0	1	2	3	4	5	6
0	true	false	false	false	false	false	false
1	true						
2	true						
3	true						

subset[i][j] is true iff a subset of the first ***i*** items sums up to ***j***

The Subset Sum dynamic programming solution

```
set  = [1, 2, 3]
```

`subset[n+1][sum+1]`

i\j	0	1	2	3	4	5	6
0	true	false	false	false	false	false	false
1	true						
2	true						
3	true						

go left by set[i-1]

OR

The Subset Sum dynamic programming solution

```
set  = [1, 2, 3]
```

`subset[n+1][sum+1]`

i\j	0	1	2	3	4	5	6
0	**true**	**false**	**false**	**false**	**false**	**false**	**false**
1	**true**						
2	**true**						
3	**true**						

go left by set[i-1]

OR

Subset sum bottom-up dynamic programming

```
boolean SSS(int set[], int sum, int n) {
    boolean[][] subset = new boolean[n+1][sum+1];
    //easy cases
    for (int i = 0; i <= n; i++) subset[i][0] = true;
    for (int i = 1; i <= sum; i++) subset[0][i] = false;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {

            if (j >= set[i-1])
                subset[i][j] = subset[i-1][j] ||
                               subset[i–1][j-set[i-1]];
            else subset[i][j] = subset[i-1][j];
        }
    }
    return subset[n][sum];
}
```

Example 6: Edit Distance

- Given two strings
  - a string S of length n
  - a string T of length m
- find **minimum** number of **edits** to convert S to T
  - called Levenshtein Distance (LD)
- Example: "WEASEL” → "SEASHELL"
- Possible edits
  - Change a character
  - Delete a character
  - Insert a character

Edit Distance

- LD("WEASEL", "SEASHELL") = 3
  - Consider "WEASEL":
    - Change W to S
    - Add an H in position 5
    - Add an L in position 8
  - Result is SEASHELL
  - If we **reverse** the arguments, we get the (**same**) distance from T to S (but the edits may be different)
- How can we determine this?
  - We can define it in a **recursive** way **initially**
  - Then we will use **dynamic programming** to improve the run-time

Edit Distance

- We want to calculate D(S, T) where n is the length of S and m is the length of T

  If n = 0 // BASE CASE 1

  return m (m appends will create T from S)

  else if m = 0 // BASE CASE 2

  return n (n deletes will create T from S)

  else

  Consider **character n** of S and **character m** of T

  - Now we have some possibilities

Edit Distance: last characters match

If characters **match**

- Result is the same as for strings with last characters removed (since they match)
- return D(n-1, m-1)
- Recursively solve the same problem with both strings one character smaller

Edit Distance: last characters don't match

- If characters **do not match** -- more possibilities here
  - We could have a **mismatch** at that char**:**
    - Example:
      - `S = -------------X`
      - `T = -------------Y`
    - Change X to Y, then recursively solve the same problem but with both strings one character smaller
    - return D(n-1, m-1) + 1

Edit Distance: last characters don't match

- S could have an **extra** character
  - Example:
    - `S = -----------X`**`Y`**
    - `T = ------------`**`X`**
  - Delete Y, then recursively solve the same problem, with S one char smaller but with T the same size
  - return D(n-1, m) + 1

Edit Distance: last characters don't match

- S could be **missing** a character there
  - Example:
    - `S = ------------`**`Y`**
    - `T = -----------Y`**`X`**
  - Append X onto S, then recursively solve the same problem with S the original size and T one char smaller
  - return D(n, m-1) + 1

Edit Distance: recursive solution

- Unfortunately, we don't know which of these gives the minimum distance until we try them all!
- We must try all subproblems and choose the one that gives the minimum result
  - up to 3 recursive calls (mismatch case) for each original call
  - worst-case run-time: Theta(3^{n+m})

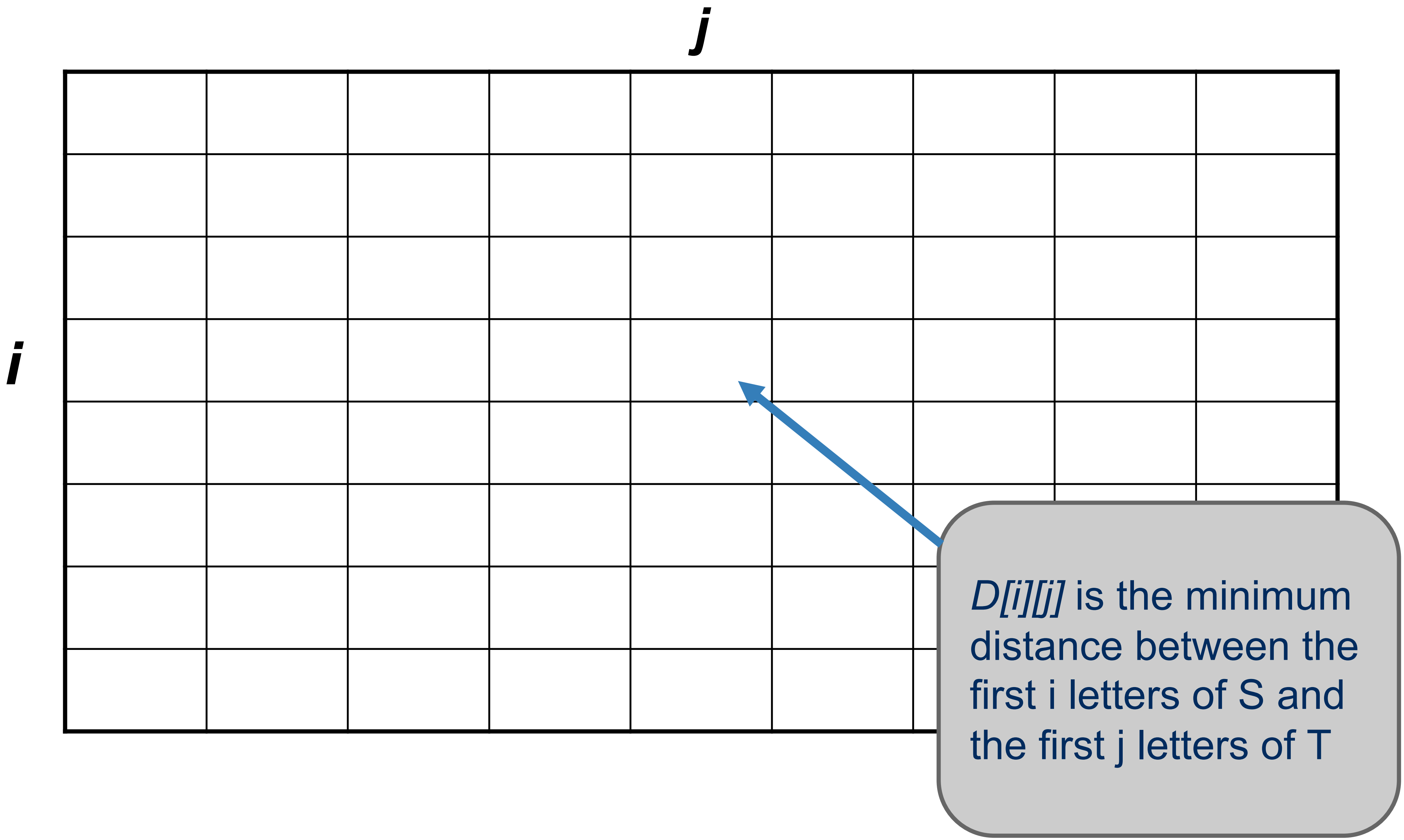
Edit Distance: dynamic programming

- How can we do this more efficiently?
  - One unique subproblem for each value of n and m
  - a two-dimensional array for all possible values for n and m
  - calculate the same D() values but bottom up rather than top down

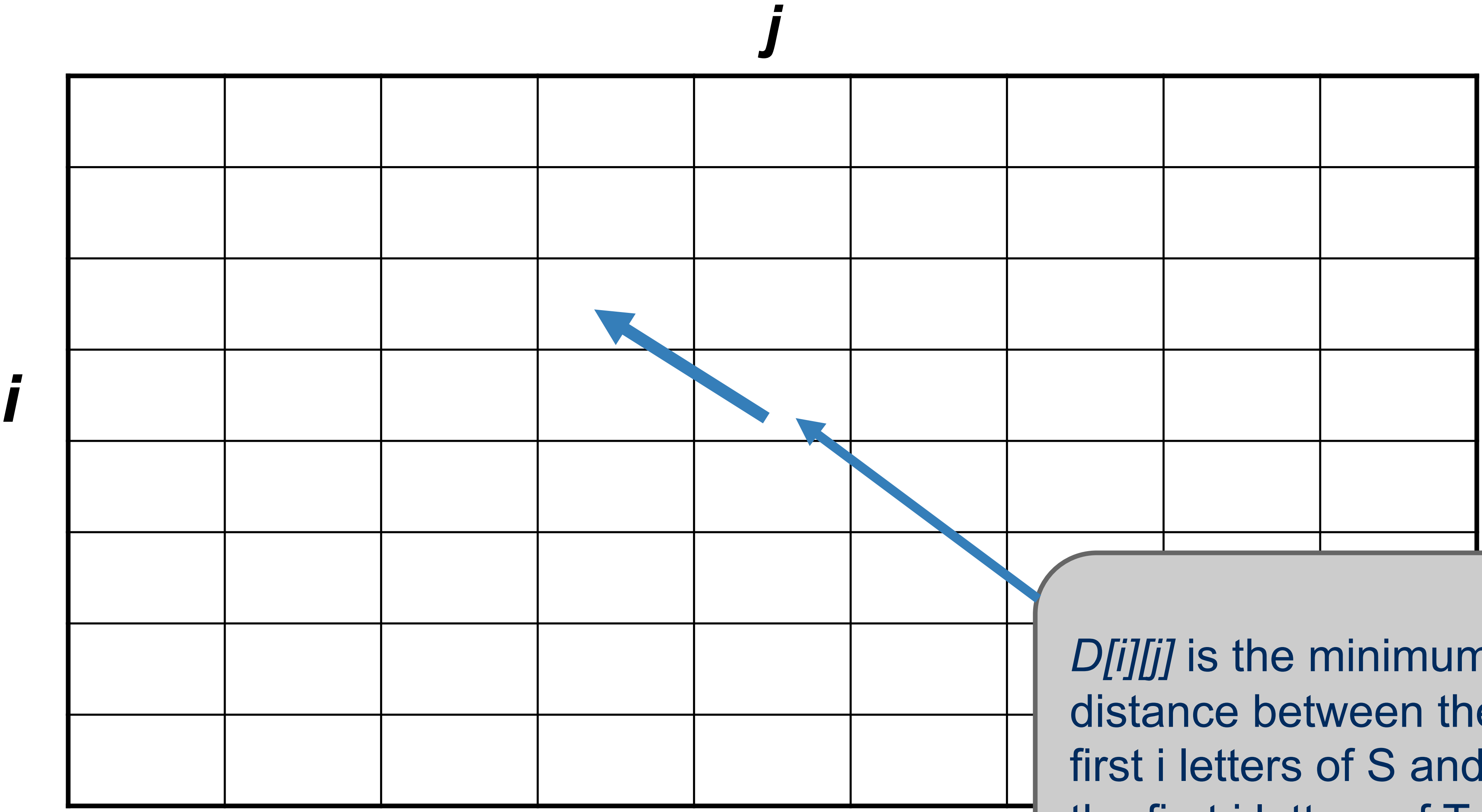
Edit Distance

- D[i, j] = D[i-1, j-1] if we have a **match**
- When we have a mismatch, **minimum** of the cells
  - D[i-1, j-1] + 1
    - Change char at this point in S
  - D[i-1, j] + 1
    - Delete a char from S
  - D[i, j-1] + 1
    - Append a char to S

Edit Distance in case of matching letters i and j

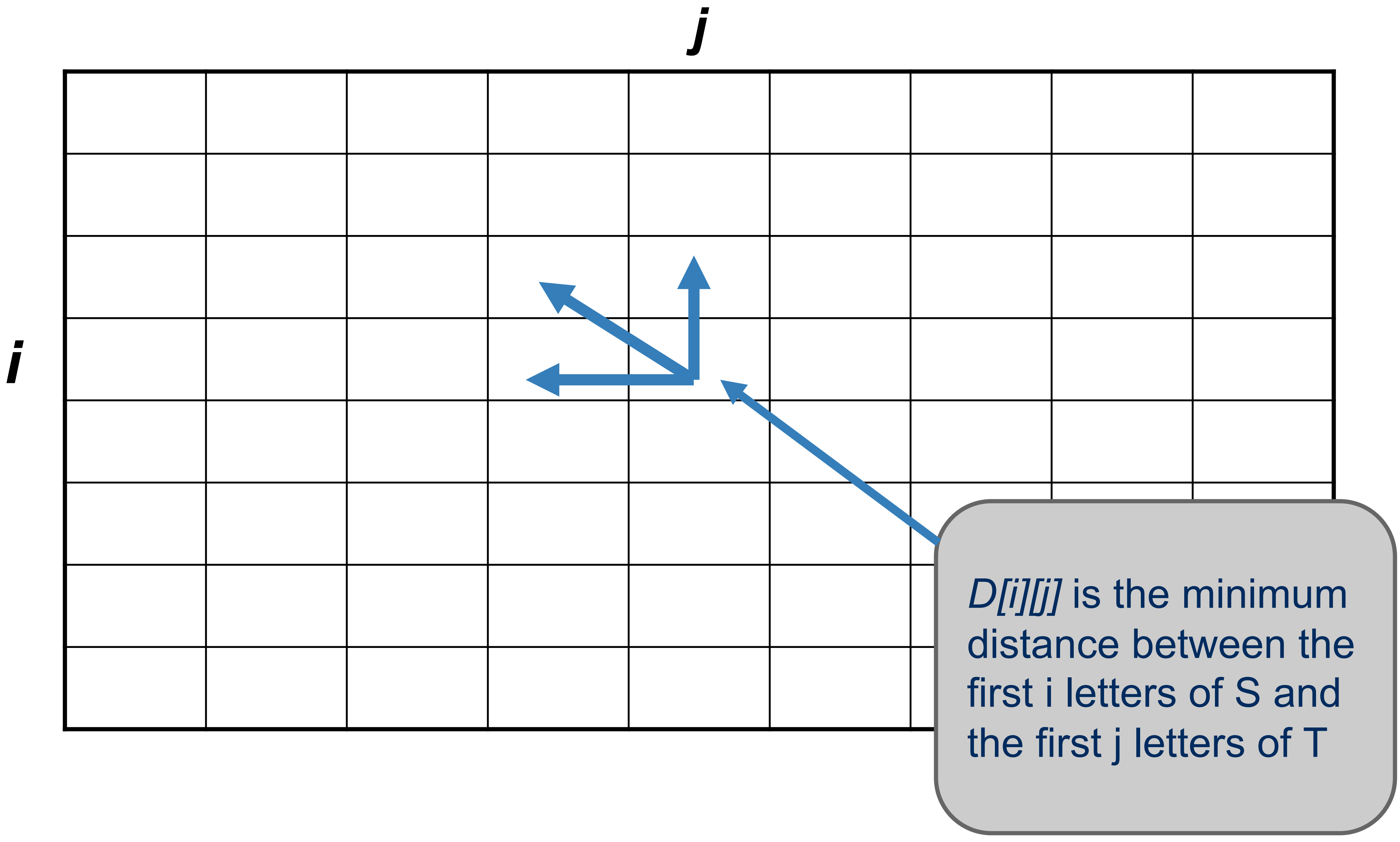


Edit Distance in case of matching letters i and j



D[i][j] is the minimum distance between the first i letters of S and the first j letters of T

Edit Distance in case of mismatching letters i and j



Edit Distance

- value at **bottom right corner** is the edit distance
- Example:
  - PROTEIN → ROTTEN

Edit Distance

		P	R	O	T	E	I	N
R								
O								
T								
T								
E								
N								

Edit Distance

		P	R	O	T	E	I	N
	0	1	2	3	4	5	6	7
R	1							
O	2							
T	3							
T	4							
E	5							
N	6							

Edit Distance

		P	R	O	T	E	I	N
	0	1	2	3	4	5	6	7
R	1	1	1	2	3	4	5	6
O	2	2	2	1	2	3	4	5
T	3	3	3	2	1	2	3	4
T	4	4	4	3	2	2	3	4
E	5	5	5	4	3	2	3	4
N	6	6	6	5	4	3	3	

Edit Distance

- This is cool!
- Run-time is Theta(mn)
  - As opposed to the 3^{n+m} of the recursive version
- Not **pseudo-polynomial** like subset sum and knapsack
- Optimized versions can reduce space from Theta(mn) to Theta(m+n)

Example 7: Longest Common Subsequence

- Given two sequences, return the **longest common subsequence**

- `Example:`
  - `A Q S R J K V B I`
    `Q B W F J V I T U`
  - `A` **`Q`** `S R` **`J`** `K` **`V`** `B` **`I`**
    `Q` `B W F` **`J V I`** `T U`

- We'll consider a **relaxation** of the problem and only look for the ***length*** of the longest common subsequence

LCS dynamic programming example

`x = A Q S R J B I`

`y = Q B I J T U T`

i\j		Q	B	I	J	T	U	T
A								
Q								
S								
R								
J								
B								
I								

LCS dynamic programming solution

```
int LCSLength(String x, String y) {
   int[][] m = new int[x.length + 1][y.length + 1];
   for (int i=0; i <= x.length; i++) {
           for (int j=0; j <= y.length; j++) {
                 if (i == 0 || j == 0) m[i][j] = 0;
                 if (x.charAt(i) == y.charAt(j))
                       m[i][j] = m[i-1][j-1] + 1;
                 else
                       m[i][j] = max(m[i][j-1], m[i-1][j]);
           }
     }
   return m[x.length][y.length];
}
```

Example 8: Reinforcement Learning

- A type of **Machine Learning**
  - an **agent** (e.g., a robot)
  - learns an optimal **policy**
  - only by getting **rewards** from the **environment**

Example

Source: https://medium.com/machine-learning-for-humans/

Input: Markov Decision Process

- A set of **states**
  - e.g., maze locations, agent health
- A set of agent **actions**
  - e.g., move left, move right, etc.
- **Probabilities** of ending up in a state given a current state and an action
  - e.g., move left action → moving left with 1.0 prob. if no wall
  - e.g., if wall, move left action → moving right or up or down with 0.33 prob.
- **Reward** function
  - depends on state and action
  - e.g., high reward for moving up from below cheese

Input: Markov Decision Process

- A set of **states**
  - think graph **vertices**
- A set of agent **actions**
  - think graph **edges**
- **Probabilities** of ending up in a state given a current state and an action
- **Reward** function
  - think edge **weights**
- A special case: all information readily available
  - called **Planning**

Output: Optimal Agent Policy

- An agent policy determines the **probability** of taking an action given a state
  - e.g., prob. 1.0 for moving left from start
- An optimal policy gives the **maximum total reward**
- Let's embed rewards into **state values**
  - think distance[] in **Bellman**-Ford
- An optimal policy gives the **maximum total state value**

Expectations

- Expected **value** of a state?
  - depends on actions
  - Sum_{all actions}:
    - prob. of action **(from policy)** * expected reward
- Expected reward from an action depends on
  - immediate reward (from reward function)
  - values of states reachable through the action
  - Sum_{all states}:
    - prob. of reaching state * **state value**

Using Dynamic Programming to Solve MDP

- Data Structure: Array of state values
- Step 0: Start with an **initial** policy and initial state values
  - e.g., all actions equally likely and state values = 0
- Step 1: Compute expected state values
  - optional: **iterate** until values **converge**
- Step 2: **Modify** policy to take the best action with probability 1.0 (given the current state values)
- Repeat Step 1 and 2 until policy **converges**